M1 Info
N-REINES

Mickaël Guillot
Sombi Rakotoniary

1er avril 2013

Table des matières

Introduction

Le problème à modéliser correspond au problèmes dit des N-reines. Il s'agit de placer N [1] reines sur un échequier de taille N*N. La contrainte principale est que deux reines ne doivent pas se menacer. Cette règles regroupes trois contraintes. Elles indiquent que 2 reines n'ont pas le droit d'être sur la même ligne, même colonne ou même diagonale. Ces règles s'appliquent à un ensemble de cases ayant une position unique et dont le contenu peut être une reine ou rien.

Dans un premier temps, nous allons voir la structure de notre programme pour bien comprendre son fonctionnement puis nous expliciterons les algorithmes.

1. nombre entier supérieur à 0

Chapitre 1

Programme

Pour développer notre programme nous avons choisi le langage C++ au travers la bibliothèque Qt. Ce choix a été fait afin de modéliser et d'afficher nos résultats avec un échequier et des reines.

1.1 Classes

Les clasees sont séparées en deux grandes parties, la première pour modéliser et afficher les résultats, la deuxième pour calculer ces mêmes résultats.

Fenêtre

Classe	Utilisation
case	modélise le domaine {0,1} qui représente le vide ou une reine
	modélise la variable unique {0,1,...,Nb} qui correspond à la position dans l'échequier
échequier	modélise et affiche le contenu de l'échequier
fenêtre	modélise et affiche le choix des algoritmes et des solutions

Algorithmes

Classe	Utilisation
algo	Utilisation
backtrack	modélise l'algorithme backtracking
forwardchecking	modélise l'algorithme forward checking
generateandtest	modélise l'algorithme generate and test
recherchelocal	modélise l'algorithme recherce locale

1.2 Utilisation

Choix de l'algorithme

Pour afficher un résultat il est nécessaire de choisir tout d'abord un algorithme de recherche. Pour cela, il faut choisir dans la liste suivante une des possibilités :

- Generate and test
- Backtracking
- Forward Checking
- Recherche locale

Afficher résultat

Une fois l'algorithme choisi, le bouton lancer permet d'afficher la première solution trouvée puis il est possible de faire défiler les solutions grâce aux boutons suivant et précédent.

Chapitre 2

Algorithmes

2.1 Generate and test

Principe

L'algorithme de Generate and test [1]consiste à générer toutes les solutions possibles sans tenir compte des contraintes imposées. L'étape suivante permet de tester les contraintes sur chaque solution trouvée et s'arrête lorsque l'on trouve une solution satisfaisante.

Avis

L'avantage de cet algorithme repose dans sa simplicité en revanche il demande de stocker tout les resultats avant les tests. Cela implique une bonne gestion de la mémoire et une estimation de la capacité à exploiter les résultats en terme de temps.

2.2 Backtracking

Principe

L'algorithme Backtracking [2] est une amélioration de generate and test. Il consiste à tester au fur et à mesure qu'on place les reines les contraintes. Si une contrainte n'est pas satisfaite alors il est inutile de continuer à placer les reines. On retourne en arrière à la dernière solution satisfaisante pour essayer une autre configuration.

Avis

Plus efficace dans les tests, cet algorithme permet d'élaguer plusieurs solutions insatisfaisantes d'un seul coup.

2.3 Forward Checking

Principe

L'algorithme Forward Checking [3] est une amélioration de Backtrack. Lors de l'attribution d'une reine à une case, les cases non satisfaisantes sont supprimés des choix possibles des futures

1. générer et tester
2. retourner en arrière
3. vérifier en avant

reines. S'il n'existe plus de choix satisfaisant les contraintes alors on retourne à la dernière solution satisfaisante pour essayer une autre configuration.

Avis

Plus efficace dans les tests, cet algorithme permet d'élaguer plus vite les solutions insatisfaisantes. Ainsi la contrainte de mémoire est moindre.

2.4 Recherche locale

La recherche locale est une méta-heuristique utilisée pour résoudre des problèmes d'optimisation difficiles. Les algorithmes de recherche locale passent d'une solution à une autre en utilisant une stratégie de recherche dans chaque voisinage jusqu'à ce qu'une solution considérée comme optimale soit trouvée. Dans notre cas à chaque voisinage (ici une colonne) on choisit la première case ayant le moins de conflit. Puis on passe au voisinage suivant jusqu'à avoir une solution viable ou qu'il n'est plus de choix possible c'est-à-dire que pour chaque voisinages il ne trouve pas de meilleur solution.

Conclusion

Ce projet a permis de constater les différences d'efficacité entre les algorithmes implémentés. Ci-dessous un tableau récapitulatif du temps de calcul et du nombre de solutions trouvées en fonction du nombre de reines et de l'algorithme utilisé. On peut constater que le plus efficace est l'algorithme de Forward Checking.

Donnees

nombre de reines	nombre de solutions	it. GT	tp. GT	it. BT	tp. BT	it. FC	tp. FC
4	2	341	0	61	0	17	0
5	10	3906	1ms	221	0	54	0
6	4	55987	24ms	895	1ms	153	0
7	40	9608000	318ms	3585	4ms	552	0
8	92	19173961	7.97	15721	23ms	2057	2ms
9	352	435848050	3.36min	72379	127ms	8394	11ms
10	724	>int	2h37	348151	743ms	35539	53ms